

POK Developer Guide

POK Team

December 2, 2023

Contents

1	About this manual	1
1.1	About POK	1
2	Coding guidelines	1
3	Source organization	2
3.1	kernel	2
3.2	libpok	3
4	Optimization (about the <code>POK_CONFIG_OPTIMIZE_FOR_GENERATED_CODE</code>)	3
5	Documentation	4
5.1	User Guide	4
5.2	Code documentation	4
6	Submit a patch	5
7	Algorithms guidelines	5
8	GDB'ing POK with QEMU	5
8.1	Troubleshooting	6
9	Commit	8
10	Join the POK developer network !	8

1 About this manual

This manual provides information about the development of POK. It indicates coding rules and naming convention so that everybody could improve POK by modifying its source code.

1.1 About POK

POK is a free operating system compliant with the ARINC653 and MILS standards. It provides time and space partitioning to isolate software in partitions. POK is released under the BSD licence and thus, could be modified and used for commercial as well as non-commercial use. To have more information about the licence of the projet, see POK website¹.

2 Coding guidelines

There are our coding guidelines:

1. Prefix for types: pok_
2. Prefix for functions : pok_
3. Prefix for macro: POK_ but not for conditional compiling:
 - (a) When the code needs a fonctionnality, we define a macro with the prefix POK_NEEDS_my-fonctionnality
 - (b) When a macro configures the kernel or user code, it has the prefix POK_CONFIG_my-config-directive
4. Indentation for ANY loop/condition
5. Commits must be as small as possible
6. Reduce machine-dependent code as more as possible
7. Each header-file must begin with `#ifdef __POK_SUBCATEGORY_FILENAME_H__`
8. Loop and condition style is :

```
condition
{
}
```

and NOT

```
condition {
}
```

9. If you introduce a new function for the userland, you must add relevant documentation in the `doc/userguide/` directory.

¹<http://pok.gunnm.org>

3 Source organization

At the root directory, two main directories are available: `kernel` and `libpok`. We detail the organization and the guidelines for each subdirectory of `kernel` and `libpok`.

3.1 kernel

In the `kernel`, sources files are supposed to contain few lines of code. In consequence, there is one file for each service.

- `arch`: contains arch-dependent code. There is one directory for each architecture and one subdirectory for each BSP. For example, files for the `x86` architecture and the `x86-qemu` BSP are located in the `arch/x86/x86-qemu` directory.
- `core`: contains the core fonctionnality of POK - threads, partitions, health monitoring,
- `include`: contains headers files. The organization of header files is the same than source files. So, you will find `core`, `middleware` or `arch` directories in the `include` directory.
- `libc`: provides some fonctionnalities for printing things. Functions located in this directory are here mainly for debugging purposes.
- `middleware`: contain the code for inter-partitions communication (sampling and queueing ports). It also contains some fonctionnalities about virtual ports routing.

3.2 libpok

In `libpok`, sources files are supposed to contain more code than in `kernel`. So, there is one file for each functions. There is the organization and purpose of each directory.

- `arch`: contains architecture dependent files. Unlike the `kernel`, there is no need to separate each BSP so there is no subdirecties for each architecture.
- `arinc653`: contains the implementation of the ARINC653 layer.
- `core`: contains the main fonctionnality of POK. It contains the thread service, lockobjects, semaphores, events.
- `drivers`: contains device drivers implemented in POK.
- `include`: contains header files. As in the `kernel`, the structure of this directory follow the structure of the sources.
- `libc`: contains the C-library of POK (`stdio`, `stdlib` and so on).
- `libm`: contains the `libmath` backported from NetBSD.
- `middleware`: contains sources for sampling and queueing ports (interfacing with the `kernel` - inter-partition communication) but also blackboard and buffers (intra-partition communication)

4 Optimization (about the `POK_CONFIG_OPTIMIZE_FOR_GENERATED_CODE`)

Systems generated with POK must be lightweight and keep a small memory footprint to be compliant with embedded requirements and ensures a good code coverage. When a system is written by hand, the libpok layer contains all its functionalities. It is more convenient for the developer, he does not have to specify which functions he needs.

However, when a system is generated from AADL models, it defines the macro `POK_CONFIG_OPTIMIZE_FOR_GENERATED_CODE` and sets its values to 1. It means that the code specifies precisely which functions are used. Then, the generated code specifies which services it needs using `POK_NEEDS*` macros. For example, the `POK_NEEDS_LIBC_STDIO` specifies that it needs all functions of `libc/stdio`.

Then, each function of libpok is surrounded with a `POK_CONFIG_NEEDS_FUNC*` or `POK_CONFIG_NEEDS_*`. **You have to introduce that in your code when you introduce new services in POK.**

Then, the file in `include/core/dependencies.h` specifies which functions are needed for each service. When the `POK_CONFIG_OPTIMIZE_FOR_GENERATED_CODE` is not set, all functions are enabled (default behavior). But is defined, functions are carefully activated, depending on their service.

5 Documentation

5.1 User Guide

Each improvement and enhancement in kernel or libpok must be documented in the userguide (see `doc/userguide` in the POK sources) to keep a consistency between the documentation and the sources.

5.2 Code documentation

The code **must be** documented using doxygen. At each release, we issue a documentation in HTML and PDF using code documentation. The following paragraphs indicate at least what information should be included in the sources **at least**. Keep in mind that the more the code is documented, the best it is for users.

Beginning of a file

Specify the file, the author, the data and a brief description. You can have an example in `kernel/core/thread.c`. For example, the following comments provide these informations. It should be located at the beginning of the file.

```
/**
 * \file    core/thread.c
 * \author  Julien Delange
 * \date    2008-2009
 * \brief   Thread management in kernel
 */
```

Functions

You **MUST** document each function and details what the function do. You specify that with a comment just before the function. The comment must begin with `/**`. There is an exemple for the function `pok_thread_init`:

```
/**
 * Initialize threads array, put their default values
 * and so on
 */
void pok_thread_init(void)
{
    ...
}
```

Global variables

Each global variable **must be** documented. As functions, you put a comment just before the global variable. This comment **must begin** with `/**`. There is an example for the global variable `pok_threads`:

```
/**
 * We declare an array of threads. The amount of threads
 * is fixed by the software developper and we add two threads
 *   - one for the kernel thread (this code)
 *   - one for the idle task
 */
pok_thread_t pok_threads[POK_CONFIG_NB_THREADS];
```

6 Submit a patch

If you found a bug or just want to send us an improvement, you can reach us at the following address: `pok-devel` at `listes dot enst dot fr`. Please send an email with the patch. We will answer and potentially merge your patch in the current version of POK.

7 Algorithms guidelines

Before introducing new functions or modifying existing ones, please qualify your code in terms of complexity, memory overhead, computation overhead, determinism. POK targets safety-critical systems, and so, its functions must provide high confidence to the user and must address these problems in its functions.

Moreover, we always follow the motto *Keep It Simple, Stupid* for each function: code must be understandable and documentation to be spread over users or developers.

8 GDB'ing POK with QEMU

POK allows you to attach a remote GDB to monitor the kernel *or* its partitions.

To do so, go to your example directory and run the system in debug mode.

```
$ cd $POK_PATH/examples/partition-threads
$ make run-gdb
```

QEMU should be paused. Now run GDB using the kernel image.

```
$ gdb generated-code/kernel/kernel.elf
...
(gdb) target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) continue
```

You're all set if you want to debug the kernel, but what if you want to instrumentate a partition?

In GDB, we first have to let the kernel know about the symbols of the partition. But we also need to know where they are loaded in kernel space. Let's say we want to debug partition #1. One way to know where it was relocated would be:

```
(gdb) p pok_partitions[0].base_addr
$1 = 1175552
```

Please note that `pok_partition_init` *must* have completed or the array won't be initialized yet.

Now we can load the symbol table with the correct offset.

```
(gdb) add-symbol-file generated-code/cpu/part1/part1.elf 1175552
add symbol table from file "generated-code/cpu/part1/part1.elf" at
      .text_addr = 0x11f000
(y or n) y
Reading symbols from /home/laurent/pok/examples/partitions-threads/generated-code/cpu/part1/pa
(gdb) b user_hello_part1
Breakpoint 1 at 0x11f17a: file ../../../../hello1.c, line 21.
(gdb) continue
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000017a in ?? ()
```

You will notice debug symbols are missing, although we loaded them above. This is because the memory mapping is not the same in kernel end userland. We have to load the symbol file again in place of the kernel.

```
(gdb) symbol-file generated-code/cpu/part1/part1.elf
Load new symbol table from "/home/laurent/pok/examples/partitions-threads/generated-code/cpu/p
Reading symbols from /home/laurent/pok/examples/partitions-threads/generated-code/cpu/part1/pa
(gdb) bt
#0  user_hello_part1 () at ../../../../hello1.c:21
#1  0xc4830845 in ?? ()
```

8.1 Troubleshooting

GDB does not break on partition symbols

In case your binaries are not generated by POK, it might happen, that GDB tells you it set a breakpoint at `user_hello_part1`, but it does not stop at this function. This happens, if the `.text` section of the binary file that you load with `add-symbol-file` does not start at the beginning of the file. If `.text` has an offset of `0x1000` you need to add this offset to the address passed to `add-symbol-file` (see `(gdb) help add-symbol-file` for details)

```
(gdb) p/x pok_partitions[0].base_addr
$1 = 0x31c000
(gdb) p/x pok_partitions[0].base_addr + 0x1000
$2 = 0x31d000
(gdb) add-symbol-file generated-code/cpu/part1/part1.elf 0x31d000
...
```

Where exactly the `.text` section starts can be figured out with

```
$ readelf -S generated-code/cpu/part1/part1.elf
There are 22 section headers, starting at offset 0x2039b0:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00001000	001000	01db18	00	AX	0	0	16
[2]	.init	PROGBITS	0001eb18	01eb18	00000d	00	AX	0	0	1
[3]	.fini	PROGBITS	0001eb25	01eb25	000008	00	AX	0	0	1
[4]	.rodata	PROGBITS	0001eb40	01eb40	00164c	00	A	0	0	32
[5]	.eh_frame	PROGBITS	0002018c	02018c	009df4	00	A	0	0	4
[6]	.ctors	PROGBITS	0002a000	02a000	000008	00	WA	0	0	4
[7]	.dtors	PROGBITS	0002a008	02a008	000008	00	WA	0	0	4
[8]	.jcr	PROGBITS	0002a010	02a010	000004	00	WA	0	0	4
[9]	.data	PROGBITS	0002a020	02a020	000a44	00	WA	0	0	32

[10]	.bss	NOBITS	0002aa80	02aa64	008ec0	00	WA	0	0	32
[11]	.comment	PROGBITS	00000000	02aa64	00008f	01	MS	0	0	1
[12]	.debug_aranges	PROGBITS	00000000	02aaf8	0028c0	00		0	0	8
[13]	.debug_info	PROGBITS	00000000	02d3b8	15f4bd	00		0	0	1
[14]	.debug_abbrev	PROGBITS	00000000	18c875	02702a	00		0	0	1
[15]	.debug_line	PROGBITS	00000000	1b389f	036d5e	00		0	0	1
[16]	.debug_str	PROGBITS	00000000	1ea5fd	011073	01	MS	0	0	1
[17]	.debug_loc	PROGBITS	00000000	1fb670	007c98	00		0	0	1
[18]	.debug_ranges	PROGBITS	00000000	203308	0005e8	00		0	0	1
[19]	.shstrtab	STRTAB	00000000	2038f0	0000bf	00		0	0	1
[20]	.symtab	SYMTAB	00000000	203d20	007520	10		21	1208	4
[21]	.strtab	STRTAB	00000000	20b240	006fad	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)

O (extra OS processing required) o (OS specific), p (processor specific)

9 Commit

Before pushing changes, please check that your test suite passes on all supported architectures and that the document is up-to-date and can be properly built.

10 Join the POK developer network !

If you want to join the POK team, please send us an email ([pok-devel at listes dot enst dot fr](mailto:pok-devel@listes dot enst dot fr)). We are always looking for developers with strong skills in C, ASM and low-level programming.

If you are interested and think you can improve the project, you're welcome!